|  |  |
| --- | --- |
| **Politechnika Świętokrzyska**  **Wydział Elektrotechniki Automatyki i Informatyki** | |
| Technologie obiektowe  Projekt | Piotr Grzyb 1ID21A  Plugin do środowiska Eclipse z wykorzystaniem narzędzia PlantUML do tworzenia diagramu klas |

1. Zagadnienie projektowe
2. Omówienie narzędzi projektowych
   1. PlantUML
   2. Wbudowane biblioteki Eclipse
   3. Java SWT
   4. Eclipse for RCP and RAP Developers
3. Schemat działania pluginu
4. Kluczowe klasy
5. Działanie pluginu i testy
6. Wnioski
7. Literatura

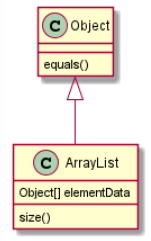
# Zagadnienie projektowe

Celem projektu było skonstruowanie pluginu do środowiska Eclipse, który przy pomocy narzędzia PlantUML generowałby diagram klas. Plugin działa głównie w oparciu   
o biblioteki wbudowane do środowiska Eclipse, które umożliwiają w prosty sposób dostęp do zasobów wspomnianego IDE.

Plugin wykorzystuje aktywny workbech i jego właściwości. Przy ich pomocy generowane są dane diagramu które następnie przetwarzane są przez PlantUML API, a kolejno trafiają do widoku pluginu i diagram rysowany jest na kanwie. Poza API, dwoma kluczowymi do działania programu bibliotekami są Eclipse.core oraz Eclipse.ui. Pierwsza pozwala, jak można się domyśleć po nazwie, dostęp do rdzenia środowiska, a druga pozwala na dwie rzeczy: tworzenie workbench’y oraz dostęp do nich. Ale o w większym szczególe te narzędzia zostaną opisane w kolejnym rozdziale.

Wiemy już co plugin generuje, diagram klas, oraz bardzo ogólnie w jaki sposób to robi. Czym więc w ogóle jest diagram klas?

Diagram klas pozwala zobrazować zbiór klas, interfejsów oraz związków między nimi. Jest to tak naprawdę graf złożony z wierzchołków (klasy, interfejsy itp.) oraz łuków (relacji między nimi). Diagram klas stanowi opis statyki całego systemu, ze szczególnością uwypukla związki między klasami.

Diagram ten zawiera oprócz klas i interfejsów, adnotacji, abstrakcji, typów enumerycznych itd. zmienne, oraz metody. Przykładowy mały diagram klas (wygenerowany przez PlantUML) wygląda następująco:

Jako że plugin wykorzystuje wbudowane biblioteki środowiska Eclipse to nie zadziała w żadnym innym środowisku. Uruchomienie go np. w InteliJ’u jest absolutnie nie możliwe. Nie będzie w nim działał.

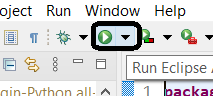
# Omówienie narzędzi projektowych

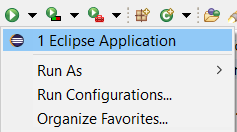
Zanim przejdziemy do omówienia kolejnych narzędzi użytych podczas tworzenia pluginu, przejdźmy do tego co potrzebne jest aby w ogóle ten projekt uruchomić.

Wersja Javy nie powinna mieć większego znaczenia, wersja środowiska Eclipse również jednak wszelkie testy wykonano na wersji 2021-03.

Aby uruchomić projekt potrzebne są następujące narzędzia:

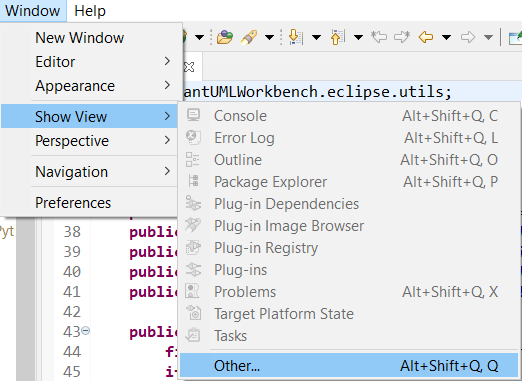
* Eclipse IDE for Java Developers
* Eclipse IDE for RCP and RAP Developers (najlepiej w tej samej wersji co poprzednik)
* Plugin PDE Tools (użyto wersji 1.2.4)

Po zainstalowaniu całego oprogramowania wystarczy uruchomić Eclipse IDE for RCP and RAP Developers, umieścić tam projekt pluginu (zarówno projekt zawierający jego działanie na worbenchu, oraz PlantUML API, bez niego projekt nie zadziała). Kolejno należy nacisnąć na strzałkę rozszerzającą wybór możliwych opcji uruchomienia.

Rozwiną nam się wszystkie możliwe opcje uruchomienia naszych rozwiązań, wystarczy wybrać opcję: Run As – Eclipse Aplication.

Następnie zostanie uruchomione środowisko Eclipse IDE for Java Dev. wraz z uruchomionym w nim widokiem zawierającym interfejs pluginu.

Jeżeli nie znajdziemy w nowo otwartym IDE działającego pluginu można go również uruchomić przy pomocy przeglądarki widoków, plugin w niej będzie nazywał się PlantUML. Gdzie znajdziemy przeglądarkę widoków? Wystarczy zerknąć na poniższy zrzut ekarnu:



Teraz wystarczy już tylko otworzyć w nowym IDE dowolny projekt w którym znajdują się jakieś elementy (klasy itp.) aby ukazał nam się diagram klas.

# PlantUML

PlantUML to narzędzie pozwalające generować różnego rodzaju diagramy. Przy pomocy tego komponentu możemy tworzyć diagramy:

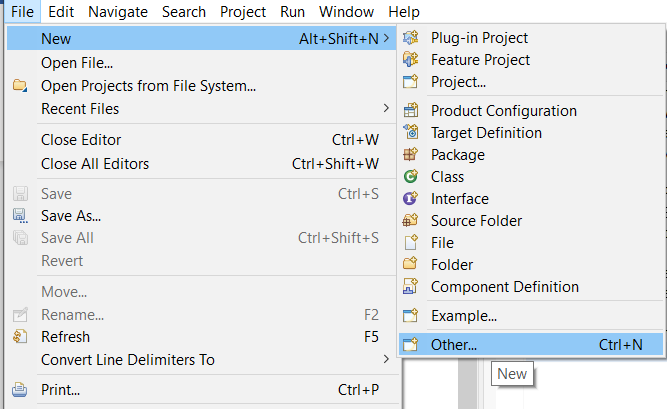
* Sekwencji
* Klas
* Przypadków
* Obiektów
* Aktywności
* Komponentów
* Stanu
* Czasów

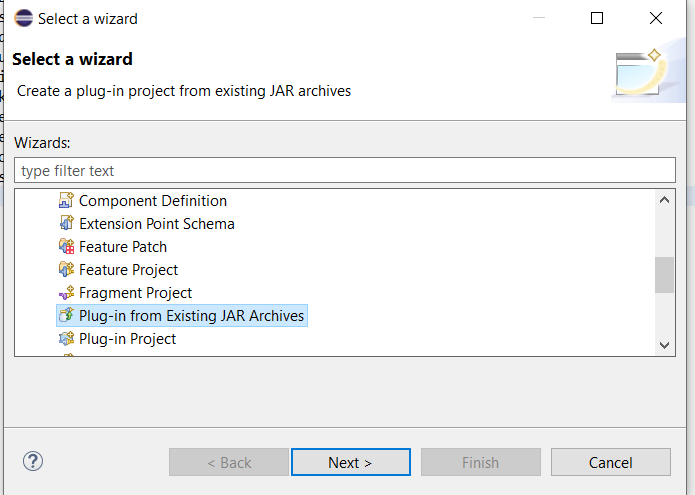
W tym projekcie użyto API udostępnianego do Javy przez PlantUML. API to pozwala generować diagramy na podstawie ciągu znaków (String). Narzędzie to ma swoją składnie która co ciekawe nie różni się specjalnie od danych dostarczanych o elementach języka Java przez wewnętrzne biblioteki Eclipse (core i ui). Dzięki temu nie jest konieczna gwałtowna modyfikacja danych które otrzymujemy. Mimo to nie oznacza to że nie jest ona w ogóle konieczna. Pakiet PlantUMLWorkbench.postprocessors dostarcza właśnie taką funkcjonalność.

Jeśli chodzi o samo PlantUML to możemy z niego korzystać również online, bezpośrednio na ich stronie, poniżej link:

<http://www.plantuml.com/plantuml/uml/SyfFKj2rKt3CoKnELR1Io4ZDoSa70000>

Aby w ogóle myśleć o użyciu powyższego API przy tworzeniu plugina, należało stworzyć plugin na podstawie pliku .jar z tym API. W przeciwnym razie nie było by możliwe wykorzystanie tej biblioteki. Możemy więc stwierdzić że tak naprawdę korzystamy z pluginu PlantUML, w którym znajduje się samo API. Aby stworzyć plugin na podstawie pliku .jar należy skorzystać z wbudowanych narzędzi Eclipse IDE for RCP and RAP Dev. Wybieramy File/New/Other. A następnie wyszukujemy Plug-in from Existing JAR Archives. Kolejno konfigurujemy plik MANIFEST.MF aby wskazać co eksportujemy i możemy spokojnie zacząć korzystać z API.



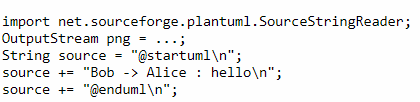


Kod pliku MANIFEST.MF:

|  |
| --- |
| Manifest-Version: 1.0  Bundle-ManifestVersion: 2  Bundle-Name: Plantuml library plug-in  Bundle-SymbolicName: PlantUMLAPI  Bundle-Version: 1.2021.5  Bundle-Vendor: PlantUML Team  Bundle-RequiredExecutionEnvironment: JavaSE-1.8  Bundle-ActivationPolicy: lazy  Export-Package: net.sourceforge.plantuml;version="1.2021.5",  net.sourceforge.plantuml.core;version="1.2021.5",  net.sourceforge.plantuml.security;version="1.2021.5" |

Nazwy eksportowanych paczek zostawiono domyślne. Samo użycie narzędzia ogranicza się tak naprawdę do zaimportowania w kilku miejscach konkretnych klas. Przykładowo poniżej 3 importy pliku DiagramData.java:

|  |
| --- |
| **import** net.sourceforge.plantuml.SourceStringReader;  **import** net.sourceforge.plantuml.StringUtils;  **import** net.sourceforge.plantuml.core.DiagramDescription; |

Przed wszystkim interesuje nas *SourecStringReader*. To właśnie ta klasa pozwala na tworzenie obrazu na podstawie tekstowych danych diagramu. Przykładowe użycie tej klasy z dokumentacji poniżej:

# Wbudowane biblioteki Eclipse

W pluginie korzystamy z kilku wbudowanych i specyficznych dla środowiska Eclpise bibilotek. A konkretnie bibliotek Eclipse.core oraz Eclipse.ui to równie ważne narzędzia potrzebne do działania pluginu. To dzięki nim możliwe jest stworzenie informacji o diagramie, bez których działanie pluginu było by niemożliwe.

Przyjrzyjmy się przede wszystkim eclipse.ui. To dzięki tej bibliotece plugin jest w stanie otrzymywać informacje o plikach znajdujących się w obszarze roboczym. Kluczowe funkcje z których korzysta plugin to między innymi:

* PlatformUI.getworkbench() – funkcja która zwraca workbench z aktywnej instancji.
* getWorkbench().getActiveWorkbenchWindow() – zwraca aktualnie otwarte okno dla workbench’a
* getCurrnetActiveWindows().getShell – zwraca shell tego workbench’a

Aby skorzystać z tych funkcji musimy zaimportować poniższe moduły:

|  |
| --- |
| **import** org.eclipse.ui.IWorkbench;  **import** org.eclipse.ui.IWorkbenchWindow;  **import** org.eclipse.ui.PlatformUI; |

Bibliotek tych możemy użyć jedynie w środowisku Eclipse, w żadnym innym. Czyni je to bibliotekami dedykowanymi.

Jeśli o pakiet eclipse.ui to w poniższym linku znajdziemy opis wszystkich dostępnych funkcji:

<http://archive.eclipse.org/eclipse/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/package-summary.html>

Jeśli chodzi o pakiet eclipse.core to informacje o nim znajdziemy poniżej:

<https://www.eclipse.org/eclipse/platform-core/documents/core_overview.html>

# Java SWT

Plugin do obsługi strony wizualnej używa biblioteki graficznej Java SWT(Standard Widget Toolkit).

Jest to biblioteka graficzna biblioteka graficzna dla środowiska Java, stanowiąca alternatywę dla standardowych bibliotek Javy: AWT i Swing. Biblioteka ta jest rozwijana na zasadach open source przez Fundację Eclipse. Ideą projektu jest, aby wygląd komponentów graficznych i ich zachowanie (standardowe skróty klawiaturowe, reakcje na działania myszy) było w pełni zgodne z komponentami typowymi dla danego systemu, a ich prędkość nie odbiegała od natywnego interfejsu. SWT jest wykorzystywana jako podstawowa biblioteka graficzna środowiska Eclipse.

W praktyce SWT to kod Java który za pomocą JNI otacza komponenty zdefiniowane na danej platformie i dostarcza interfejs wspólny dla różnych platform.

Rozwiązanie to ma zalety:

* Komponenty są zwykle bardziej wydajne w użyciu w porównaniu do rozwiązań konkurencyjnych (w zależności od zastosowań i systemu)
* Po stworzeniu biblioteki SWT dla danej platformy, zwykle nie ma problemów ze zgodnością z nowymi wersjami systemu operacyjnego (np. SWT dla Win32 działa na Windows Vista, a Swing musi mieć od nowa napisany „Look and Feel”)
* SWT istnieje dla każdej znaczącej platformy i ma mniejsze wymagania niż Swing (zwłaszcza pamięciowe)
* SWT jest prostsze w użyciu niż Swing (dyskusyjne: np. nie wymaga stosowania MVC, ale trudno tworzyć własne skomplikowane komponenty w porównaniu do Swing)
* Aplikację korzystającą z SWT można skompilować do kodu natywnego (ułatwia to rozprowadzenie oprogramowania, ponieważ program ma mniejsze wymagania (bo nie uruchamia się JVM, jest w pełni natywny) i nie potrzebuje JRE, ale przeczy to idei Javy)
* Posiada kilka zaawansowanych komponentów[[1]](https://pl.wikipedia.org/wiki/Standard_Widget_Toolkit#cite_note-1) więcej niż Swing (Tree z możliwością dodawania kolumn, Expand Bar, komponenty C\*\*\*\*, czyli zaimplementowane w SWT wersje standardowych komponentów)

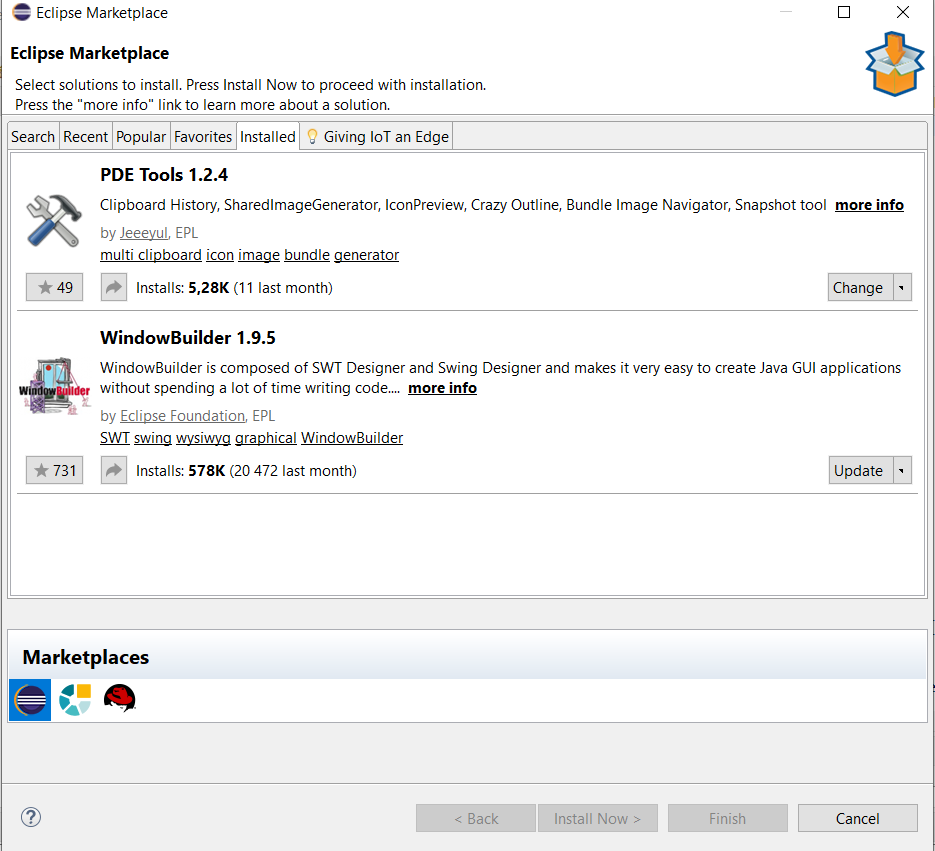
Rozwiązanie ma również wady:

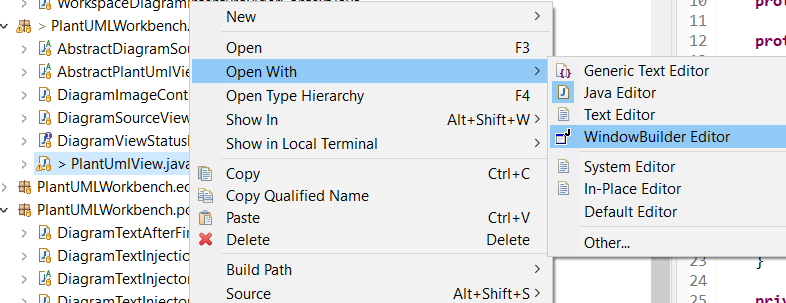
* Trzeba samemu zatroszczyć się o zwalnianie zasobów (metoda dispose())
* Rozwiązanie w większych projektach może prowadzić do bałaganu, ponieważ SWT nie wymaga korzystania z MVC, po większości obiektów SWT nie można (nie powinno się) dziedziczyć więc trudno je rozbudować (dyskusyjne: JFace daje możliwość jeszcze prostszego tworzenia okien, tabel etc. będąc wyższą warstwą abstrakcji, jednocześnie nie ograniczając dostępu do SWT)
* Potrzeba do programu w Javie dla każdej platformy dostarczyć natywną bibliotekę SWT i ewentualne jej zależności (np. libCairo pod Linuksem)
* Trzeba od nowa portować (przepisywać) wszystkie komponenty dla każdej nowej platformy (Swing jest rysowany, więc potrzeba tylko funkcji rysowania i nowej implementacji L&F, z drugiej strony wydaje się, że SWT szybciej powstaje na nowe platformy niż Swing się do nich adaptuje)
* SWT jest mocno uzależnione od platformy
* Trudno rozbudować komponent SWT lub stworzyć własny
* SWT długo nie miało odpowiednika Rendererów i wciąż trudno tworzyć tabelki czy listy z zawartością graficzną

Wewnątrz projektu biblioteka używana jest przed wszystkim do stworzenia kanwy na której umieszczane będą diagramy, oraz za późniejsze kontrolowanie narysowanego diagramu. Możliwe jest jego powiększanie, pomniejszanie, zapisywanie itp.

Większość kodu wykorzysującego bibliotekę SWT znajdziemy w pakiecie *imagecontrol.*

Ponadto większość kodu została nie napisana ręcznie a stworzona przy pomocy pluginu ułatwiającego pracę z SWT – WindowBuilder 1.9.5. Narzędzie to ułatwia budowanie widoków i okienek. Współpracuje on zarówno z biblioteką SWT jak i biblioteką Swing. Na poniższych zrzutach ekranu zostały umieszczone wszystkie pluginy używane do stworzenia projektu, oraz prezentacja jak używać WindowBuilder’a.





# Eclipse for RCP and RAP Developers

Aby stworzyć plugin do konkretnego środowiska programistycznego (IDE). Początkowo projekt miał powstać dla środowiska PyCharm, jednak stopień trudności i niewielka ilość literatury na ten temat odstraszyła mnie od wyboru tego rozwiązania.

Wybór padł więc na zasugerowane przez prowadzącego środowisko Eclipse. Aby tworzyć pluginy w tym środowisku konieczne jest ściągnięcie specjalnej wersji tego też środowiska – Eclipse for RCP and RAP Developers.

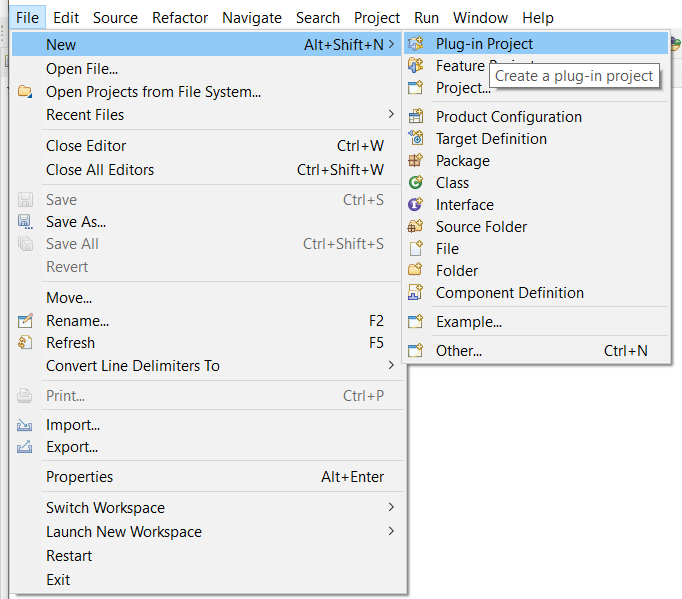
Aby jednak przetestować pluginy należy również zainstalować Eclipse for Java Developers.

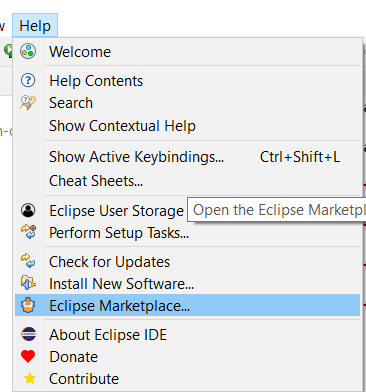
Aby zainstalować obie wersje tego oprogramowania należy udać się na stronę :

<https://www.eclipse.org/downloads/packages/installer>

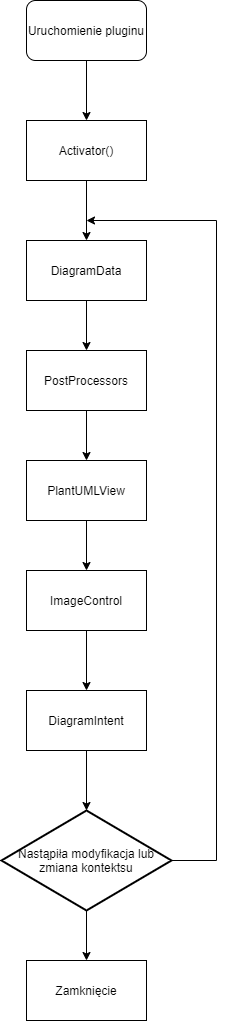
i podążać za instrukcjami.

Po zainstalowaniu obu środowisk jesteśmy prawie gotowi do działania. Przydatnym pluginem o którym wspomniałem wcześniej jest PDE Tools (Plugin Development Tools). To on umożliwia nam opcje testowania pluginów na działającym środowisku. (Run As / Eclipse Application).

W Eclipse for RCP and RAP Developers możemy tworzyć już pluginy, aby tego dokonać wystarczy uruchomić IDE i wybrać z menu przycisk File, następnie New i Plug-in Project

Aby zainstalować wspomniany wcześniej plugin – PDE Tools, należy otworzyć Eclipse Marketplace i tam go wyszukać. Poniżej zrzut przedstawiający jak dostać się do tego narzędzia.

# Schemat działania pluginu

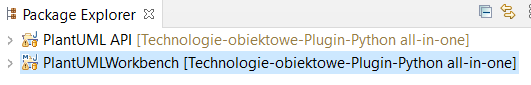


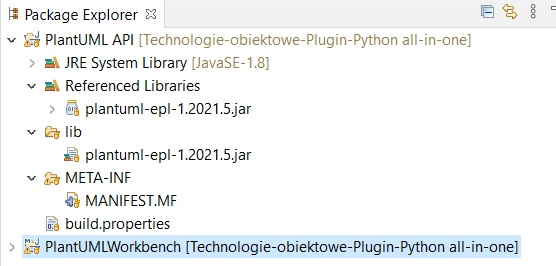
Plugin działa aktywnie według wyrysowanego diagramu. Activator zajmuje się uruchomieniem wszystkich słuchaczy i obsługą otwarcia i zamknięcia pluginu. Klasa DiagramData zawiera jak można się domyśleć dane diagramu, klasa ta wykorzystuje również klasę WorkbenchUtill, która zapewnia obsługę okna roboczego. Zanim dane diagramu trafią do okna widoku przechodzą jeszcze przez PostProcessor’y. Tam wyrzucane są niechciane znaki nowej linii i zamieniane niektóre znaki tak aby tekst diagramu odpowiadał składni PlantUML.

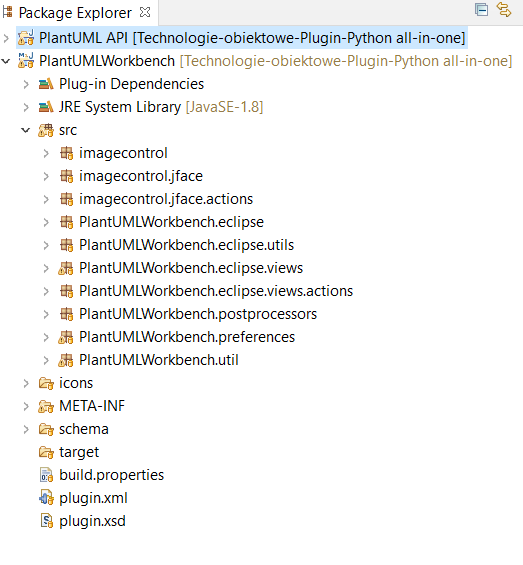
Gdy te procesory zakończą swoją pracę dane trafiają już do klasy PlantUMLView, która reprezentuje sam widok, a więc jest swego rodzajem sercem programu. To tam wyświetlane jest absolutnie wszystko i to tam znajduje się obsługa menu. Kontrole obrazu diagramu zapewniają pakiety ImageControl (powiększanie, pomniejszanie, dopasowanie do okna, reset itp.). W końcu klasa DiagramIntent i jej pochodne zajmują się sprawdzeniem czy nie nastąpiła jakaś zmiana w oknie roboczym, pliku na podstawie którego obecnie wyświetlany jest diagram itp. Dane do tego dostarcza nam ponownie biblioteka Eclipse.ui.

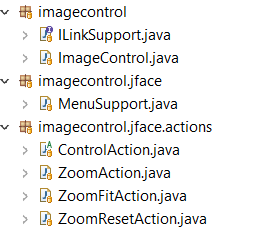
Plugin działa w takiej właśnie swego rodzaju pętli. Klasa DiagramIntent ma zapewnić poprawę wydajności, a jednocześnie zadbać o to żeby aktualizacje diagramu następowały tylko wtedy kiedy rzeczywiście nastąpiła jakaś zmiana. W rzeczywistości zdarza się jednak że plugin nie wychwytuje zmian co do sekundy. Zdarzają mu się opóźnienia. Z tego powodu możemy zawsze odświeżyć diagram wykorzystując naciśnięcie lewego przycisku myszy.

# Kluczowe klasy

Zanim przyjrzymy się poszczególnym klasą spójrzmy na ogólną budowę pluginu.

Przede wszystkim jak możemy zaobserwować plugin to tak naprawdę nie jeden pojedynczy plugin a dwa pluginy. PlantUML API musi być pluginem, ponieważ w innej formie nie moglibyśmy korzystać z API. Zawartość tego pluginu jest jednak mocno ograniczona.

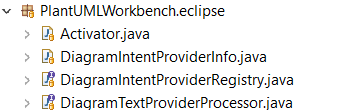
Przejdźmy zatem do właściwego pluginu. Najpierw omówmy działanie i funkcjonalność poszczególnych pakietów.

Pakiet ImageControl i jego składowe:

Pakiet ten zgodnie z nazwą zajmuje się kontrolą obrazu. Chodzi tu o wyrysowany już gotowy obraz diagramu. Dzięki temu pakietowi mamy takie opcje jak:

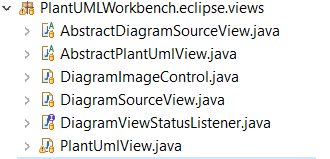
* Powiększanie diagramu
* Zmniejszanie diagramu
* Reset zoomu
* Dopasowanie zoomu do okna widoku

Kolejny pakiet to PlantUMLWorkbench.eclipse



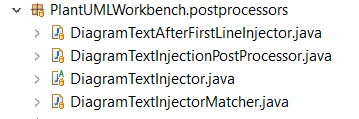
Pakiet ten zawiera przed wszystkim aktywator. Aktywator pozwala w ogóle uruchomić plugin i go zamknąć. To on uruchamia również większość słuchaczy który generują tekst diagramu, sprawdzają jego poprawność itp. Pozostałe elementy to mało znaczące interfejsy.

Podpakiet poprzedniego pakietu eclipse.view



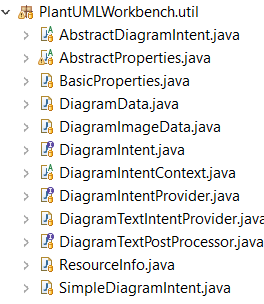
To zdecydowanie najważniejszy pakiet z całego projektu. Klasa PlantUmlView to klasa która odpowiada za sam widok w którym dzieje się absolutnie wszystko. Znajduje się tu również klasa DiagramSourceView która sprawia że możliwe jest działanie opcji skopiowania tekstu diagramu po kliknięciu na niego prawym przyciskiem myszy. Znajduje się tu także słuchacz statusu widoku diagramu.

Kolejny pakiet zawierający postprocessory:

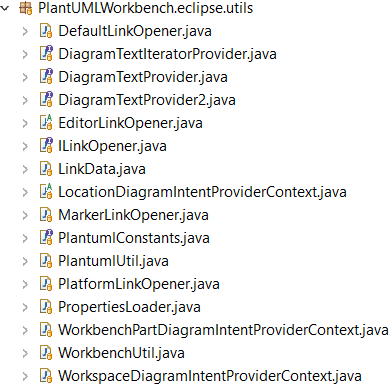


W tym pakiecie możemy dostosować tekst diagramu do formatu PlantUML. Pierwsza klasa tego pakietu wprowadza zmiany w pierwszej linii i dostosowuje ją do formatu wcześniej wspomnianego (@startuml). DiagramTextInjectorMatcher pozwala na zamianę kilku fragmentu ciągu znaków na format PlantUML’owy.

Kolejny pakiet:



Pakiet narzędzi, zawiera przeróżne klasy z których korzystają praktycznie wszystkie pakiety. Znajdziemy tutaj klasę DiagramData która zapewnia dane do diagramu. Ponadto znajdziemy tu implemencje DiagramIntent. Oprócz tego możem tu znaleźć procesory wspomagające prace innych klas.

Ostatni pakiet

W pakiecie tym znajduje się przewszystkim WorbenchUtil, który ułatwia współpracę całego pluginu z oknem roboczym. Ponadto znajdują się tu wszystkie klasy po prostu nie pasowały do innych pakietów.

Klasa WorkbenchUtil

|  |
| --- |
| **package** PlantUMLWorkbench.eclipse.utils;  **import** org.eclipse.jface.dialogs.MessageDialog;  **import** org.eclipse.swt.widgets.Shell;  **import** org.eclipse.ui.IWorkbench;  **import** org.eclipse.ui.IWorkbenchWindow;  **import** org.eclipse.ui.PlatformUI;  **public** **class** WorkbenchUtil {  **public** **static** IWorkbenchWindow getCurrentActiveWindows() {  **return** *getWorkbench*().getActiveWorkbenchWindow();  }  **public** **static** IWorkbench getWorkbench() {  **return** PlatformUI.*getWorkbench*();  }  **public** **static** Shell getShell() {  **return** *getCurrentActiveWindows*().getShell();  }  **public** **static** **void** errorBox(String message) {  *errorBox*("ERROR", message, **null**);  }  **public** **static** **void** errorBox(String title, String message, Throwable e) {  MessageDialog.*openError*(*getShell*(), title, message + "\n"  + (e != **null** ? e.getMessage() : ""));  }    **public** **static** **void** errorBox(String message, Throwable e) {  *errorBox*("ERROR", message, e);  }  } |

Klasa ta korzysta z elementów biblioteki eclpise.ui, aby zapewnić pluginowi prostą współpracę z oknem roboczym. Korzystamy z obiektu PlatformUI, aby otrzymać nasze okno robocze przy pomocy metody getWorkbech(). Aby otrzymać otwarte okno używamy metody getWorkbench().getActiveWorkbenchWindow().

Ponadto klasa ta umożliwia dostęp do shell’a okna roboczego przy wykorzystaniu metody getActiveWorkbenchWindow().getShell().

Klasa ta udostępnia także zarządzanie błędami wyrzucanymi przez okno robocze, przy pomocy metody errorBox.

|  |
| --- |
| **package** PlantUMLWorkbench.util;  **import** java.io.ByteArrayInputStream;  **import** java.io.ByteArrayOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.nio.charset.StandardCharsets;  **import** java.util.Collection;  **import** java.util.Map;  **import** java.util.regex.Matcher;  **import** java.util.regex.Pattern;  **import** org.eclipse.core.resources.IResource;  **import** org.eclipse.core.resources.ResourcesPlugin;  **import** org.eclipse.core.runtime.IPath;  **import** org.eclipse.jface.preference.IPreferenceStore;  **import** org.eclipse.swt.graphics.ImageData;  **import** org.eclipse.swt.graphics.Rectangle;  **import** net.sourceforge.plantuml.FileFormat;  **import** net.sourceforge.plantuml.FileFormatOption;  **import** net.sourceforge.plantuml.FileSystem;  **import** net.sourceforge.plantuml.OptionFlags;  **import** net.sourceforge.plantuml.SourceStringReader;  **import** net.sourceforge.plantuml.StringUtils;  **import** net.sourceforge.plantuml.core.DiagramDescription;  **import** PlantUMLWorkbench.eclipse.Activator;  **import** PlantUMLWorkbench.eclipse.utils.LinkData;  **import** PlantUMLWorkbench.eclipse.utils.PlantumlConstants;  **import** PlantUMLWorkbench.eclipse.utils.WorkbenchUtil;  **import** net.sourceforge.plantuml.security.SFile;  **public** **class** DiagramData {  // diagram source  **private** **final** String textDiagram;  // number of images, derived from textDiagram  **private** **int** imageCount;  // the file from which the diagram was derived/extracted, used as context for resolving includes  **private** IPath original;  // attributes from file marker  **private** Map<String, Object> markerAttributes;  **private** **static** **final** Pattern ***pattern*** = Pattern.*compile*("(?i)(?m)^\\W\*newpage( .\*)?$");  **public** DiagramData(**final** String diagramText) {  textDiagram = diagramText;  imageCount = 1;  // We must count the number of "newpage" instructions  **final** Matcher matcherNewpage = ***pattern***.matcher(diagramText);  **while** (matcherNewpage.find()) {  imageCount++;  }  }  **public** String getTextDiagram() {  **return** textDiagram;  }  **public** **int** getImageCount() {  **return** imageCount;  }  **public** **void** setOriginal(**final** IPath original) {  **this**.original = original;  }  **public** IPath getOriginal() {  **return** original;  }  **public** Map<String, Object> getMarkerAttributes() {  **return** markerAttributes;  }  **public** **void** setMarkerAttributes(**final** Map<String, Object> markerAttributes) {  **this**.markerAttributes = markerAttributes;  }  **public** ImageData getImage(**final** **int** imageNum, **final** Collection<LinkData> links) {  // generate the image for textDiagram and imageNumber  **if** (original != **null**) {  // find the real file, which may be linked and thus is not located under the root itself  **final** IResource member = ResourcesPlugin.*getWorkspace*().getRoot().getFile(original);  **final** SFile dirPath = SFile.*fromFile*(member.getLocation().toFile().getAbsoluteFile().getParentFile());  FileSystem.*getInstance*().setCurrentDir(dirPath);  } **else** {  FileSystem.*getInstance*().reset();  }  OptionFlags.*getInstance*().setQuiet(**true**);  **return** *getImage*(textDiagram, imageNum, links);  }  **public** ImageData getImage() {  **return** getImage(0, **null**);  }  **private** **static** **void** setGraphvizPath() {  **final** IPreferenceStore prefStore = Activator.*getDefault*().getPreferenceStore();  **final** String dotPath = prefStore.getString(PlantumlConstants.***GRAPHVIZ\_PATH***);  **if** (dotPath != **null** && (! "".equals(dotPath))) {  System.*setProperty*("GRAPHVIZ\_DOT", dotPath);  }  }  **private** **static** FileFormatOption *imageFileFormatOption* = **new** FileFormatOption(FileFormat.***PNG***);  **private** **static** ImageData getImage(**final** String textDiagram, **final** **int** imageNum, **final** Collection<LinkData> links) {  *setGraphvizPath*();  ImageData imageData = **null**;  **try** (ByteArrayOutputStream os = **new** ByteArrayOutputStream()) {  // image generation  **final** SourceStringReader reader = **new** SourceStringReader(textDiagram);  **final** DiagramDescription desc = reader.outputImage(os, imageNum, *imageFileFormatOption*);  **if** (links != **null**) {  **final** String cMapData = reader.getCMapData(0, *imageFileFormatOption*);  **if** (cMapData != **null**) {  *parseImageMapString*(cMapData, links);  }  }  os.flush();  **if** (desc != **null** && StringUtils.*isNotEmpty*(desc.getDescription())) {  **try** (InputStream is = **new** ByteArrayInputStream(os.toByteArray())) {  imageData = **new** ImageData(is);  }  }  } **catch** (**final** IOException e) {  WorkbenchUtil.*errorBox*("Error during image generation.", e);  }  **return** imageData;  }  **private** **static** **void** parseImageMapString(**final** String cMapData, **final** Collection<LinkData> links) {  **final** String[] areaElements = cMapData.split(Pattern.*quote*("<area "));  **for** (**final** String areaElement : areaElements) {  **final** String coords = *getAttributeValue*(areaElement, "coords");  **if** (coords != **null**) {  **final** LinkData link = **new** LinkData();  link.href = *getAttributeValue*(areaElement, "href");  link.title = *getAttributeValue*(areaElement, "title");  link.altText = *getAttributeValue*(areaElement, "alt");  **final** String[] ints = coords.split(",");  **if** (ints.length == 4) {  **try** {  **final** **int** x1 = Integer.*valueOf*(ints[0]), y1 = Integer.*valueOf*(ints[1]), x2 = Integer.*valueOf*(ints[2]), y2 = Integer.*valueOf*(ints[3]);  link.rect = **new** Rectangle(x1, y1, x2 - x1, y2 - y1);  links.add(link);  } **catch** (**final** NumberFormatException e) {  }  }  }  }  **if** (links != **null** && links.size() > 0) {  *scaleLinks*(links);  }  }  // assumes image is scaled, but the margin is the same  **private** **static** **void** scaleLinks(**final** Collection<LinkData> links) {  **int** minX = Integer.***MAX\_VALUE***, minY = Integer.***MAX\_VALUE***;  **for** (**final** LinkData link : links) {  **if** (link.rect.x < minX) {  minX = link.rect.x;  }  **if** (link.rect.y < minY) {  minY = link.rect.y;  }  }  // empirically identified  **final** **int** marginX = 7, marginY = 7;  **final** **int** sx = minX / marginX, sy = minY / marginY;  **for** (**final** LinkData link : links) {  link.rect.x /= sx;  link.rect.y /= sy;  link.rect.width /= sx;  link.rect.height /= sy;  }  }  **private** **static** String getAttributeValue(**final** String element, **final** String attributeName) {  **final** String prefix = attributeName + "=\"";  **int** start = element.indexOf(prefix);  **if** (start >= 0) {  start += prefix.length();  **final** String suffix = "\"";  **final** **int** end = element.indexOf(suffix, start);  **if** (end > start) {  **return** element.substring(start, end);  }  }  **return** **null**;  }  } |

Klasa DiagramData zajmuje się przede wszystki przechowywaniem danych diagramu i generowaniem samego diagramu. W atrybutach przechowuje dane. Oprócz standardowych metod get, które pozwalają wywłaszczyć konkretne atrybuty mamy również inne.

Korzystając z metody getImage (z biblioteki swt.graphics) generujemy kolejne diagramy na podstawie atrybutów znajdujących się w samej klasie. Do generowania diagramu służy przed wszystkim metoda z API PlantUML – StringSourceReader.

|  |
| --- |
| **package** PlantUMLWorkbench.eclipse.views;  **import** java.util.function.Supplier;  **import** org.eclipse.core.runtime.IProgressMonitor;  **import** org.eclipse.jface.action.IAction;  **import** org.eclipse.jface.action.IContributionManager;  **import** org.eclipse.jface.action.IToolBarManager;  **import** org.eclipse.swt.SWT;  **import** org.eclipse.swt.events.KeyAdapter;  **import** org.eclipse.swt.events.KeyEvent;  **import** org.eclipse.swt.widgets.Composite;  **import** org.eclipse.swt.widgets.Display;  **import** org.eclipse.swt.widgets.Event;  **import** org.eclipse.swt.widgets.Listener;  **import** org.eclipse.swt.widgets.Shell;  **import** org.eclipse.swt.widgets.TabFolder;  **import** org.eclipse.swt.widgets.TabItem;  **import** imagecontrol.ILinkSupport;  **import** imagecontrol.ImageControl;  **import** imagecontrol.jface.MenuSupport;  **import** PlantUMLWorkbench.eclipse.utils.LinkData;  **import** PlantUMLWorkbench.eclipse.utils.PlantumlConstants;  **import** PlantUMLWorkbench.eclipse.views.actions.CopyAction;  **import** PlantUMLWorkbench.eclipse.views.actions.CopyAsciiAction;  **import** PlantUMLWorkbench.eclipse.views.actions.CopySourceAction;  **import** PlantUMLWorkbench.eclipse.views.actions.ExportAction;  **import** PlantUMLWorkbench.eclipse.views.actions.PrintAction;  **import** PlantUMLWorkbench.eclipse.views.actions.SaveAction;  **import** PlantUMLWorkbench.util.DiagramData;  **import** PlantUMLWorkbench.util.DiagramImageData;  **public** **class** PlantUmlView **extends** AbstractPlantUmlView **implements** ILinkSupport {  **private** DiagramImageControl[] imageControls;  **private** TabFolder tabFolder = **null**;  **private** MenuSupport menuSupport;  @Override  **protected** **void** createDiagramControl(**final** Composite parent) {  ensureImageControls(0);  menuSupport = **new** MenuSupport();  addCanvasActions();  }  **private** **final** Listener mouseWheelListener = **new** Listener() {  @Override  **public** **void** handleEvent(**final** Event e) {  **if** ((e.stateMask & SWT.***CTRL***) != 0) {  **if** (e.count > 0) {  zoomInAction.run();  } **else** {  zoomOutAction.run();  }  }  }  };  **private** **final** KeyAdapter keyListener = **new** KeyAdapter() {  @Override  **public** **void** keyPressed(**final** KeyEvent e) {  **switch** (e.character) {  **case** '+':  zoomInAction.run();  **break**;  **case** '-':  zoomOutAction.run();  **break**;  **default**:  **break**;  }  }  };  **private** DiagramImageControl createImageControl(**final** Composite parent) {  **final** DiagramImageControl imageControl = **new** DiagramImageControl(parent);  imageControl.addLinkSupport(**this**);  menuSupport.addImageControl(imageControl);  imageControl.addListener(SWT.***MouseWheel***, mouseWheelListener);  imageControl.addKeyListener(keyListener);  **return** imageControl;  }  **private** **synchronized** **void** ensureImageControls(**final** **int** count) {  **if** (imageControls != **null** && imageControls.length == count) {  **return**;  }  **if** (tabFolder != **null** && count <= 1) {  tabFolder.dispose();  tabFolder = **null**;  } **else** **if** (imageControls != **null**) {  // dispose the imageControls that aren't needed  **final** **int** retained = (count > 1 && tabFolder == **null** ? 0 : count);  **for** (**int** i = imageControls.length - 1; i >= retained; i--) {  **if** (tabFolder != **null**) {  tabFolder.getItem(i).dispose();  }  imageControls[i].dispose();  }  } **else** {  imageControls = **new** DiagramImageControl[count];  }  // here imageControls exists, may contain disposed items and  // may be too short or long  **if** (imageControls.length != count) {  **final** DiagramImageControl[] newImageControls = **new** DiagramImageControl[count];  System.*arraycopy*(imageControls, 0, newImageControls, 0,  Math.*min*(imageControls.length, newImageControls.length));  imageControls = newImageControls;  }  **if** (count > 1 && tabFolder == **null**) {  tabFolder = **new** TabFolder(composite, SWT.***BOTTOM***);  }  // add the extra controls that are needed  **for** (**int** i = 0; i < imageControls.length; i++) {  **if** (imageControls[i] == **null** || imageControls[i].isDisposed()) {  imageControls[i] = createImageControl(tabFolder != **null** ? tabFolder : composite);  **if** (tabFolder != **null**) {  **final** TabItem tab = **new** TabItem(tabFolder, SWT.***NONE***);  tab.setText(String.*valueOf*(i + 1));  tab.setControl(imageControls[i]);  }  }  }  }  **private** **void** addCanvasActions() {  **final** Supplier<DiagramData> diagramDataSupplier = () -> diagramData;  **final** Supplier<DiagramImageData> diagramImageDataSupplier = () -> getCurrentImageControl().getDiagramImageData();  **final** Display display = composite.getDisplay();  menuSupport.addMenuAction(**new** CopyAction(diagramImageDataSupplier, display));  menuSupport.addMenuAction(**new** CopySourceAction(diagramDataSupplier, display));  menuSupport.addMenuAction(**new** CopyAsciiAction(diagramImageDataSupplier, display));  **final** Shell shell = composite.getShell();  menuSupport.addMenuAction(**new** SaveAction(diagramImageDataSupplier, shell));  menuSupport.addMenuAction(**new** ExportAction(diagramImageDataSupplier, shell));  menuSupport.addMenuAction(**new** PrintAction(diagramImageDataSupplier, shell));  }  **private** IAction zoomInAction, zoomOutAction, fitCanvasAction, showOriginalAction;  **private** **final** **float** ZOOMIN\_RATE = 1.1f; //  **private** **final** **float** ZOOMOUT\_RATE = 0.9f; //  **protected** DiagramImageControl getCurrentImageControl() {  **int** imageNum = 0;  **if** (tabFolder != **null**) {  imageNum = tabFolder.getSelectionIndex();  }  **return** imageControls.length > imageNum ? imageControls[imageNum] : **null**;  }  @Override  **protected** **void** contributeToActionBars() {  **super**.contributeToActionBars();  **final** IToolBarManager toolBarManager = getViewSite().getActionBars().getToolBarManager();  addZoomActions(toolBarManager);  }  **protected** **void** addZoomActions(**final** IContributionManager toolBarManager) {  addActions(toolBarManager, zoomInAction, zoomOutAction, fitCanvasAction, showOriginalAction);  }  /\*\*  \* Passing the focus request to the viewer's control.  \*/  @Override  **public** **void** setFocus() {  **final** DiagramImageControl imageControl = getCurrentImageControl();  (imageControl != **null** ? imageControl : composite).setFocus();  }  //  @Override  **protected** **void** updateDiagram(**final** DiagramData diagramData, **final** IProgressMonitor monitor) {  // load all images before updating controls  **final** DiagramImageData[] diagramImageDatas = **new** DiagramImageData[diagramData.getImageCount()];  **for** (**int** imageNum = 0; imageNum < diagramImageDatas.length; imageNum++) {  **if** (monitor != **null** && monitor.isCanceled()) {  **return**;  }  **final** DiagramImageData diagramImageData = **new** DiagramImageData(diagramData, imageNum, **null**);  diagramImageData.getImage();  diagramImageDatas[imageNum] = diagramImageData;  }  **if** (monitor != **null** && monitor.isCanceled()) {  **return**;  }  setDiagramViewStatus(ViewStatus.***DIAGRAM\_VIEW\_DATA***, diagramImageDatas);  asyncExec(() -> {  // check if the view still matches the model  **if** (shouldUpdateView(diagramData)) {  ensureImageControls(diagramImageDatas.length);  **for** (**int** imageNum = 0; imageNum < imageControls.length; imageNum++) {  **if** (imageControls[imageNum] != **null** && (! imageControls[imageNum].isDisposed())) {  imageControls[imageNum].updateDiagramImage(diagramImageDatas[imageNum]);  }  }  setDiagramViewStatus(ViewStatus.***DIAGRAM\_VIEW***, diagramImageDatas);  }  });  asyncExec(layoutComposite);  }  @Override  **public** Object getLink(**final** **int** x, **final** **int** y) {  **final** DiagramImageControl imageControl = getCurrentImageControl();  **if** (imageControl != **null**) {  **for** (**final** LinkData linkData : imageControl.getDiagramImageData().getLinks()) {  **if** (linkData.rect != **null** && linkData.rect.contains(x, y)) {  **return** linkData.title != **null** ? linkData.title : linkData.href;  }  }  }  **return** **null**;  }  @Override  **public** **void** openLink(**final** Object href) {  **final** DiagramImageControl imageControl = getCurrentImageControl();  **if** (imageControl != **null**) {  **for** (**final** LinkData linkData : imageControl.getDiagramImageData().getLinks()) {  // don't use equals, we want the same linkData instances as above  **if** (linkData.title == href || linkData.href == href) {  **if** (openLink(linkData)) {  **return**;  }  }  }  }  }  } |

# Działanie pluginu i testy

Plugin przede wszystkim jak już wielokrotnie zostało wspomniane generuje diagram klas, przy pomocy narzędzia PlantUML. Program jednak wykorzystuje również inne funkcje dostarczane przez API:

* Kopiowanie diagramu
* Kopiowanie źródła (tekstu na podstawie którego generowany jest diagram)
* Kopiowanie jako ASCII
* Eksportowania jako obraz
* Zapisywania
* Drukowania

Ponadto sam widok pluginu ma następujące opcje:

* Przybliżanie diagramu
* Oddalanie
* Przesuwanie w pionie i poziomie
* Dopasowania rozmiaru do rozmiaru okienka
* Odpięcia i przypięcia pluginu od okna roboczego

Uruchamianie pluginu zostało już omówione w rozdziale 2 stąd też tutaj skupię się na samym działaniu.

Plugin generuje diagram klas dla dwóch przypadków:

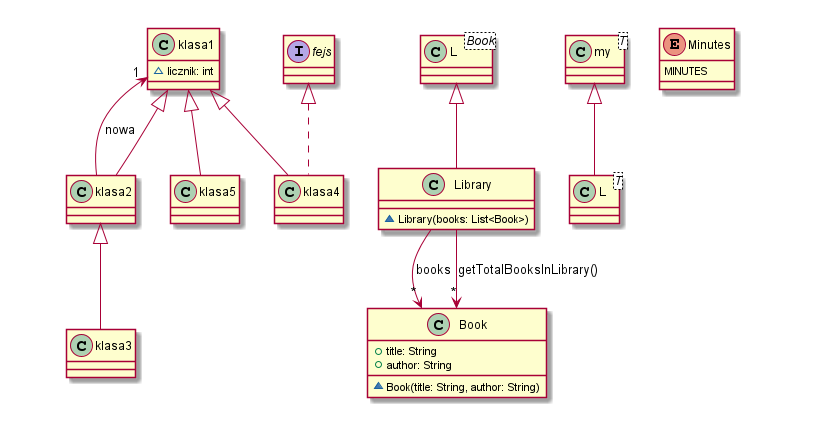
* Dla aktualnie przeglądanych plików w oknie roboczym
* Po naciśnięciu w workbench’u na pakiet

Rozważmy najpierw pierwszy przypadek. Dla niego zostało napisany prosty plik w którym testujemy różne związki między klasami (dziedziczenie, referencje itp.)

Kod klasy testowej

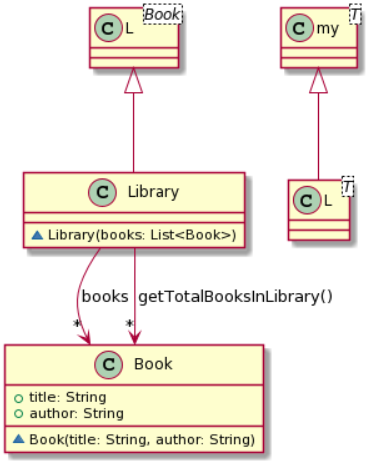
|  |
| --- |
| package src;  import java.util.List;  public class klasa1 {  int licznik;  }  class klasa2 extends klasa1 {  klasa1 nowa = new klasa1();    }  class klasa3 extends klasa2{    }  class klasa4 extends klasa1 implements fejs{    }  class klasa5 extends klasa1{  }  interface fejs{    }  class Book {    public String title;  public String author;    Book(String title, String author)  {    this.title = title;  this.author = author;  }  }  class my<T>  {    }  class L<T> extends my<T>{    }  class Library extends L<Book> {    private final List<Book> books;    Library(List<Book> books) { this.books = books; }    public List<Book> getTotalBooksInLibrary()  {    return books;  }  }  enum Minutes {  MINUTES  } |

Zrzut ekranu z działania pluginu dla klasy testowej:



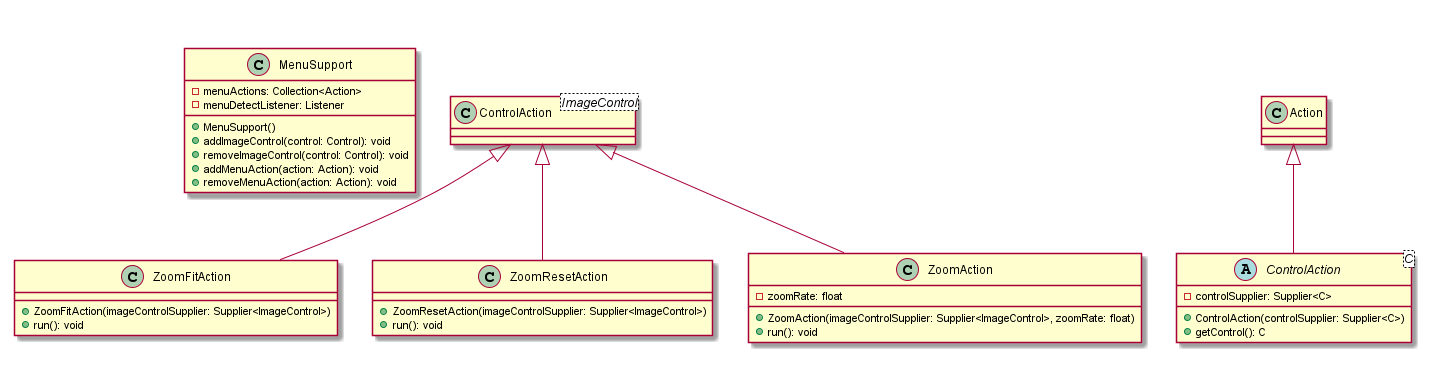
DiagramData, tekst na podstawie którego wygenerowany został diagram:

|  |
| --- |
| @startuml  class klasa1 [[java:src.klasa1]] {  ~licznik: int  }  class klasa2 [[java:src.klasa2]] {  }  class klasa1 {  }  klasa2 --> "1" klasa1 : nowa  class klasa1 {  }  klasa1 <|-- klasa2  class klasa3 [[java:src.klasa3]] {  }  class klasa2 {  }  klasa2 <|-- klasa3  class klasa4 [[java:src.klasa4]] {  }  class klasa1 {  }  klasa1 <|-- klasa4  interface fejs {  }  fejs <|.. klasa4  class klasa5 [[java:src.klasa5]] {  }  class klasa1 {  }  klasa1 <|-- klasa5  interface fejs [[java:src.fejs]] {  }  class Book [[java:src.Book]] {  +title: String  +author: String  ~Book(title: String, author: String)  }  class "my<T>" as my\_T\_ [[java:src.my]] {  }  class "L<T>" as L\_T\_ [[java:src.L]] {  }  class "my<T>" as my\_T\_ {  }  my\_T\_ <|-- L\_T\_  class Library [[java:src.Library]] {  ~Library(books: List<Book>)  }  class Book {  }  Library --> "\*" Book : books  class Book {  }  Library --> "\*" Book : getTotalBooksInLibrary()  class "L<Book>" as L\_Book\_ {  }  L\_Book\_ <|-- Library  enum Minutes [[java:src.Minutes]] {  MINUTES  }  @enduml |

Jak możemy zaobserwować wszystko działa zgodnie z założeniami. Jedynym mankamentem jest to że dla typów generycznych, plugin rysuje poszczególne obiekty osobno. Jest to jednak najprawdopodobniej wada narzędzia PlantUML, ponieważ tak wygląda diagram wygenerowany przez PlantUML po umieszczeniu w nim diagramu w formie tekstowej.

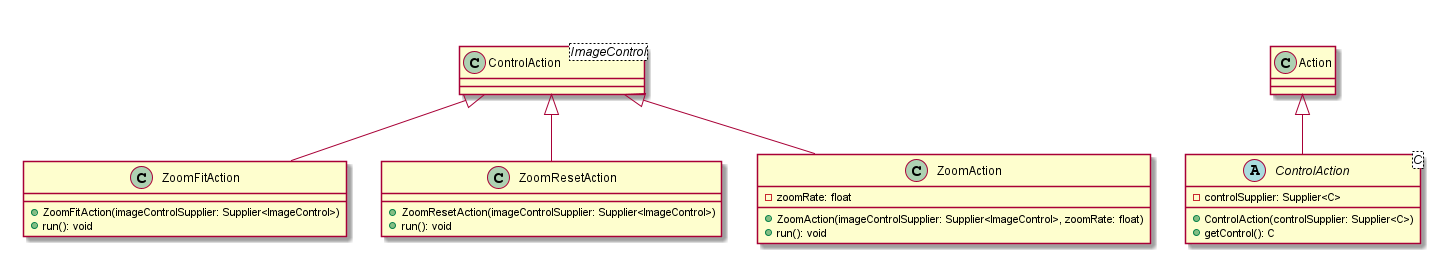
Jak widać tutaj również zależność również rysuje się osobno. PlantUML oferuje też możliwość przejrzenia wcześniej wygenerowanych już diagramów, oto link do tego diagramu:

<http://www.plantuml.com/plantuml/png/ZP8zRy8m48RdvbUSCa7R8F714Y64AdL02br89uIGIGqr98vaEwtI3tvxu_YWDbvOehlpS--REtic5H6gAl8WoucKS6o-f0-hrO5ya5YAh6SgwpLy1tUdd6LVd1rZO5m5ltRFmEiPw1uNwhS5vADe2c4_F9_4mCjFujCOJdwY29riPkZP3dtRWMLh2GrjPkG9ZOpG3Nyq2XXNLEn8Hc573xAbTD0mlHuojixOCnlVR3Q-DhEaTEu8FPNbqGPqRgxoGJ6LqnYMIZ2-Rt9IgVTI_1TEckqur2EuKFVY4nRr99s6G2GKzIRTs9v5xVnIc9p9v0fCRCwNr4ScQfUXkn5Ds5OGKTjQMC9dYrbdsumauuQNQgB7ctRTJMaBPCtpl0_DGGnjwutidggqL2JNgPppitltiW5ZZalGiPaEOvpFT0MKLmKi6AyKTUuQIsR2nVpbDNrUDXepoj-g8ly3>

Teraz spójrzmy na działanie w przypadku operowaniu na pakietach. W celu sprawdzenia tej opcji sprawdźmy jeden z pakietów samego projektu. Wystarczy otworzyć go w IDE w którym uruchomiliśmy plugin, a następnie w przeglądarce solucji nacisnąć na sam pakiet. Tak wygląda diagram klas dla pakietu imagecontrol.jface.

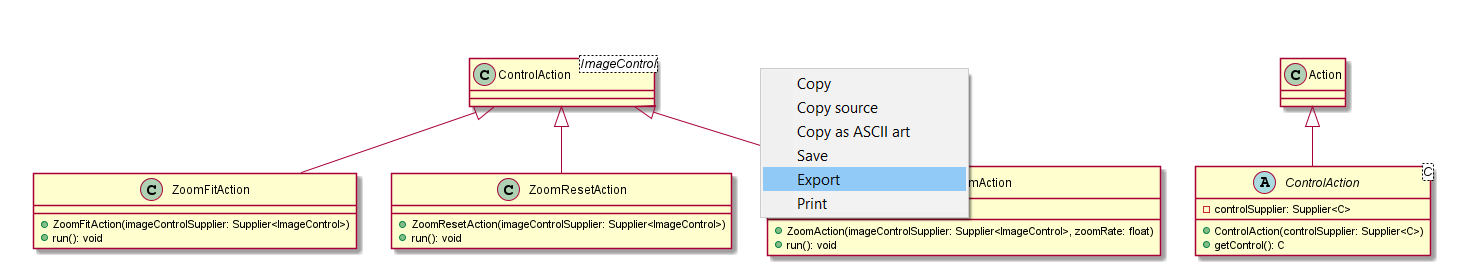
A oto tekst na podstawie którego został wygenerowany:

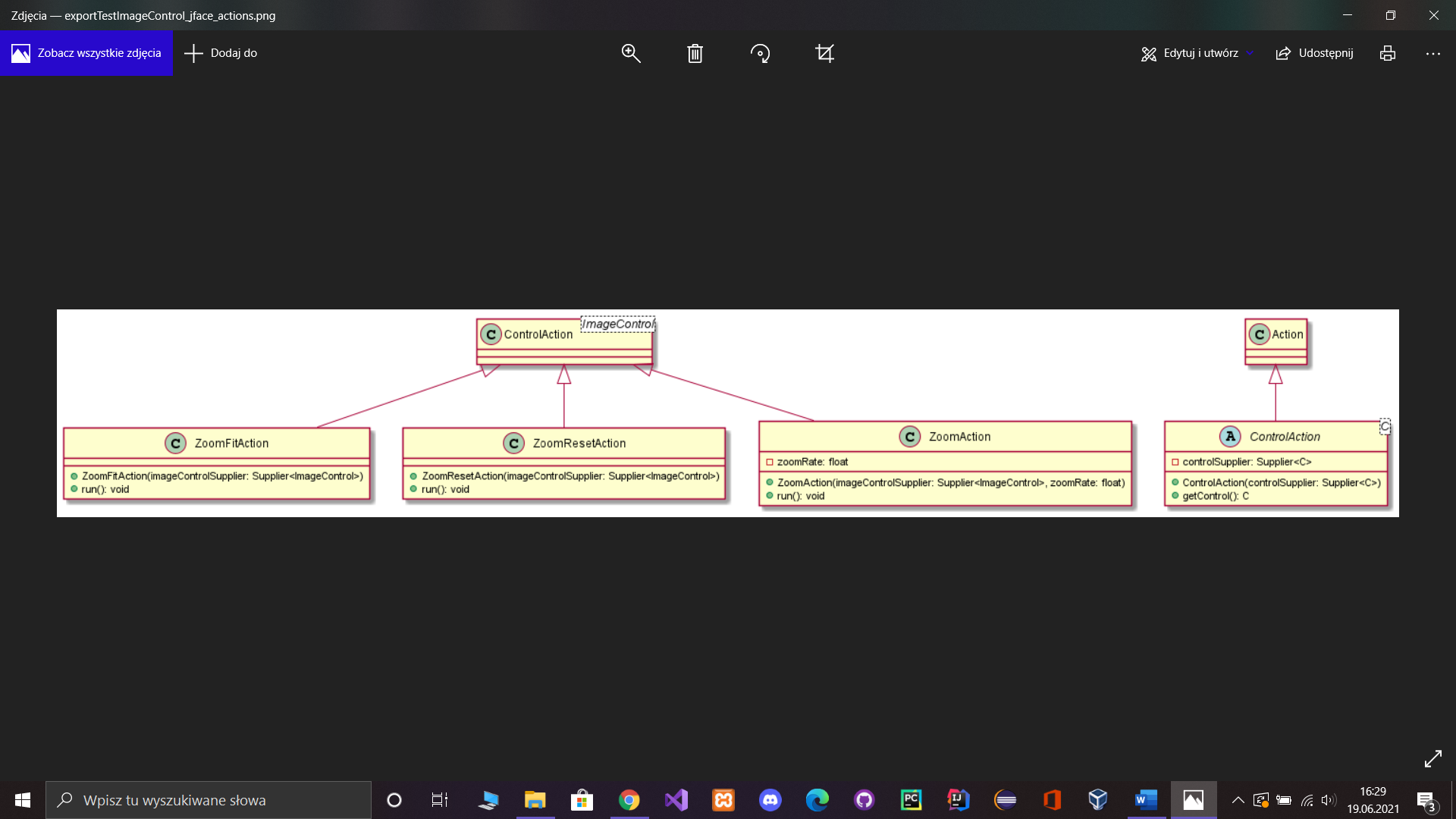
|  |
| --- |
| @startuml  class MenuSupport [[java:imagecontrol.jface.MenuSupport]] {  -menuActions: Collection<Action>  -menuDetectListener: Listener  +MenuSupport()  +addImageControl(control: Control): void  +removeImageControl(control: Control): void  +addMenuAction(action: Action): void  +removeMenuAction(action: Action): void  }  class ZoomFitAction [[java:imagecontrol.jface.actions.ZoomFitAction]] {  +ZoomFitAction(imageControlSupplier: Supplier<ImageControl>)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomFitAction  class ZoomResetAction [[java:imagecontrol.jface.actions.ZoomResetAction]] {  +ZoomResetAction(imageControlSupplier: Supplier<ImageControl>)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomResetAction  abstract class "ControlAction<C>" as ControlAction\_C\_ [[java:imagecontrol.jface.actions.ControlAction]] {  -controlSupplier: Supplier<C>  +ControlAction(controlSupplier: Supplier<C>)  +getControl(): C  }  class Action {  }  Action <|-- ControlAction\_C\_  class ZoomAction [[java:imagecontrol.jface.actions.ZoomAction]] {  -zoomRate: float  +ZoomAction(imageControlSupplier: Supplier<ImageControl>, zoomRate: float)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomAction  @enduml |

Tutaj przykładowe działanie dla pakietu imagecontrol.jface.actions:

A oto diagram w formie tekstowej:

|  |
| --- |
| @startuml  class ZoomFitAction [[java:imagecontrol.jface.actions.ZoomFitAction]] {  +ZoomFitAction(imageControlSupplier: Supplier<ImageControl>)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomFitAction  class ZoomResetAction [[java:imagecontrol.jface.actions.ZoomResetAction]] {  +ZoomResetAction(imageControlSupplier: Supplier<ImageControl>)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomResetAction  abstract class "ControlAction<C>" as ControlAction\_C\_ [[java:imagecontrol.jface.actions.ControlAction]] {  -controlSupplier: Supplier<C>  +ControlAction(controlSupplier: Supplier<C>)  +getControl(): C  }  class Action {  }  Action <|-- ControlAction\_C\_  class ZoomAction [[java:imagecontrol.jface.actions.ZoomAction]] {  -zoomRate: float  +ZoomAction(imageControlSupplier: Supplier<ImageControl>, zoomRate: float)  +run(): void  }  class "ControlAction<ImageControl>" as ControlAction\_ImageControl\_ {  }  ControlAction\_ImageControl\_ <|-- ZoomAction  @enduml |

Na tym samym przykładzie przetestujmy eksportowanie pliku.



Jak możemy zaobserwować zapis do formatu .png przebiegł pomyślnie.

# Wnioski

Plugin działa poprawnie. Udało się zrealizować założenia projektowe. Mimo to sam projekt był stosunkowo toporny i ciężki w realizacji. Dużą trudnością był sam początek prac nad projektem. Tworzenie pluginów było i wciąż stosunkowo jest nowością. Ponadto pierwsze plany stworzenia tego projektu zakładały użycie refleksji (biblioteka reflection), oraz biblioteki graphviz. Okazało się jednak że refleksje nie są tutaj optymalnym rozwiązaniem, być może spowodowane jest to tym że pierwszy raz zgłębiałem również ten temat.

Mimo trudności projekt udało się zrealizować. Plugin miał mieć również swojego brata, który konstruował by diagramy na podstawie podanej przez użytkownika ścieżki. Realizacja tego okazała się jednak zbyt czasowo wymagająca i razem z próbą nauki refleksji straciłem około 2-3 tygodni prac.

Jeśli zaś chodzi o możliwe rozwinięcie tematu to najbardziej oczywiste było by właśnie tworzenie diagramów pakietów, po wskazaniu ścieżki do nich. Czegoś w rodzaju dokumentacji. Realizacje tego na pewno ułatwił by doclet do java docs który tworzy takie diagramy. Poniżej link do strony wyjaśniającej jego działanie wraz ze sposobem instalacji:

<https://github.com/talsma-ict/umldoclet>

Podsumowując projekt, jest to świetny plugin wykorzystujący narzędzie PlantUML. Nie jest to oczywiście najbardziej zawansowane na rynku narzędzie do generowania diagramów UML, jest on jednak funkcjonalny.

# Literatura

* <https://plantuml.com/api>
* Helion - Eclipse 4. Programowanie wtyczek na przykładach
* <https://help.eclipse.org/2021-06/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/swt.htm>
* <https://help.eclipse.org/2021-06/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/swt/package-summary.html>
* <https://www.eclipse.org/windowbuilder/>
* <http://archive.eclipse.org/eclipse/downloads/documentation/2.0/html/plugins/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/package-summary.html>
* <https://www.eclipse.org/eclipse/platform-core/documents/core_overview.html>